5. Java

5.1 Einführung

5.1.1 Warum Java ?

- Kleine Programmeinheiten verteilt im Netz statt großer monolithischer Anwendungen
- *active contents* für das WWW
- Defizite bei plattformneutralen Entwicklungswerkzeugen
- Nachteile höherer Programmiersprachen

5.1.2 Vorgeschichte von Java

- Ausgangspunkt von Java: Forschungsprojekt zur Entwicklung einer Programmiersprache für vernetzte und eingebettete Anwendungen (OAK)
- ca. 5 Jahre Entwicklungsdauer; entwickelt auf der Basis verfügbarer OO-Programmiersprachen (C++, SmallTalk, Eiffel) und mit Hinblick auf weitverbreitete Standards (C++)

5.1.3 Nachteile höherer Programmiersprachen

- heutige 4GL sind im wesentlichen ca. 25 Jahre alt (C, C++, Pascal etc.)
- Hardwareinkompatibiliät und mangelnde Portabilität
- unzureichende oder nicht implementierte Standards

5.2 Grundkonzepte von Java im Überblick

5.2.1 Einfachheit

- Wesentlicher Anspruch: Einfachheit verglichen mit den Nachteilen von C++.
- Java als das bessere C++ für C-Programmierer?
- reduzierte Sprachdefinition gegenüber C++ reduziert, typische Charakteristika von C++
- fehlen
- Einfache Migration, da syntaktisch eng an C++ angelehnt
- klassisches Beispiel: Deklaration der Klasse **HelloWorld**

```
class HelloWorld
{
    static public void main(String args[])
    {
        System.out.println(“Hello, world!");
    }
}
```

5.2.2 Objektorientierung in Java

- Objektorientierung: Heute quasi-Standard bei Programmiersprachen
- Umsetzung der Objektorientierung analog zu C++ mit Vereinfachungen
- Polymorphismus
- Bildung von Unterklassen (subclassing) im wesentlichen wie in C++, aber keine Mehrfachvererbeung
- Dynamisches Binden von Klassen nach Bedarf
- Java-Basisumgebung stellt getestete Basisklassen zur Verfügung: Datentypen, I/O, Netzinterface, User Interface Toolkit (awt), jeweils nutzerseitig erweiterbar und adaptierbar
- Wie in C++: unterschiedliche Konstruktoren für eine Klasse

- Methoden: Zustand definiert durch Instanzenvariablen, Instanzenvariablen und Methoden werden in einer Klassendefinition festgelegt; Objekte werden unter Verwendung des Klassennamens instantiiert.
- Programmierung mit Feindifferenzierung der Methoden durch Zugangsklassen
- Zusätzlich: sog. finalizer, (vergleichbar in etwa mit den Dekonstruktoren in C++)
- Wie in C++ Bildung abstrakter Superklassen, die nicht unmittelbar zu Objekten instantiiert werden: Effizienzsteigerung durch einmalige Definition abstrakter Methoden in der abstrakten Oberklasse, die dann von allen “konkreten" Objekten der Unterklassen benutzt werden können (Beispiel: **Graphical - Rectangle, Circle** etc)

5.2.3 Robustheit

- Striktes *compile time* und *run time checking*
- Typprüfung zur Laufzeit erhöht die Verläßlichkeit /vereinfacht die Portabilität
- Kein Zeigerdatentyp, d.h. unwillentliches Überschreiben von Speicherbereichen weitgehend ausgeschlossen
- Einfaches Speicherverwaltungsmodell ohne Zeigertypen vermeidet klassische Programmierfehler

5.2.4 Portabilität und Architekturneutralität

- Java generiert *Bytecode* als maschinenunabhängiges Zwischenformat
- Dieselbe Folge von Java Bytecode läuft auf unterschiedlichen Plattformen
- Laufzeitumgebung: “hypothetische Java-Maschine" (*Java virtual machine - jvm*), die durch den Java-Interpreter und das Java run-time-System tatsächlich realisiert wird.
- Umfang und Größe von Datentypen und Operatoren sind eindeutig und plattformunabhängig definiert
- Alle Datentypen haben auf allen Systemen den gleichen Darstellungsbereich.

5.2.5 *Interpreter und Dynamisches Laden*

- Interpreter arbeitet auf der Basis einer Java virtual machine (plattformspezifische Implementierung)
- dynamisches Nachladen einzelner Klassen
- Zur dynamischen Erweiterung der Objektstruktur stehen sog. Interfaces zur Verfügung, die im Prinzip lediglich spezifizieren, welche Methoden ein bestimmtes Objekt benutzen kann, und die damit explizite Mehrfachvererbung ersetzen können.

5.2.6 *Multithreading*

- Threads: mehrere Ausführungsstränge gleichzeitig
- Unterschiedliche Prioritäten, Gruppierung und Synchronisation ist möglich
- Wichtig bei der Entwicklung und Realisierung interaktiver und multimedialer Programme
- Wissenschaftliche Vorarbeit: C.A.R. Hoare, System Cedar/Mesa (Xerox Parc)

5.2.7 *Speicherverwaltung*

- keine explizite Speicherallokation und -freigabe wie in C++ erforderlich
- automatic garbage collection: thread mit geringer Priorität im Hintergrund
- Speicherfreigabe bei Bedarf
- Verlieren Objektreferenzen den Skopus, kann die Speicherverwaltung erkennen, daß der für die Objektreferenz verwendete Speicherplatz wieder freigegeben werden kann.

5.2.8 *Sicherheitsaspekte*

- Mehrschichtiges Sicherheitssystem
- Speicherverwaltung: Die Speicherbelegung für ein bestimmtes Programm wird erst zur Laufzeit festgelegt, nicht bereits durch den Compiler
- Keine Pointer
- Aufwendiger Prozeß der Bytecode-Verifikation:

- Formatüberprüfung der Datensequenz
- einfacher Theorembeweiser, der überprüft, ob Zugangsrestriktionen verletzt sind und Objekte nur gemäß ihrer Definition interpretiert und nie anders
- Ergebnis der Verifikation sichert folgendes ab:
- Kein over-/underflow auf dem Stack
- Parametertypen aller Instruktionen sind korrekt
- Alle Objektzugriffe sind zulässig (private, public oder protected)
- Java Sandbox für Applets im WWW:
- Unterschiedliche Zugriffsrechte in Abhängigkeit von
- Programmtyp
- Programmort
- Identifikationsmerkmalen
- zugestandenen Rechten
- Viewersoftware/Browser

5.2.9 *Aufbau des Java-Basissystems im JDK*

Klassenbibliotheken mit verschiedenen Methoden:

Java.lang	Basisdefinition der Programmiersprache Java - Datentypen, Operatoren, Objects, Threads, Wrappers etc. Wird für jedes Java-Programm benötigt.
Java.io	Ströme (streams) und Filemanipulation (wie in anderen 4GLs üblich)
Java.net	Unterstützung zur Netzprogrammierung (sockets, telnet-Interface etc.)
Java.util	Datenstrukturen (Hash-Tabellen, Stacks), Datums- und Zeitmanipulation
java.awt	(*abstract windowing toolkit*): Klassensammlung zur Spezifikation plattformneutraler graphischer Benutzerschnittstellen (Events, Controls etc.)
java.applet	Hilfsklassen zur Applet-Programmierung

Weitere (neuere) Bibliotheken im JDK:

java.beans	Hilfsklassen für die komponentenorientierte Programmierung
java.math	Zusatzklassen für große Zahlenwerte
java.security	Hilfsklassen für Kryptographie und Sicherheit
java.rmi	remote method invocation - verteilte Programmierung
java.sql	Datenbankschnittstelle (relationale Datenbanken)
java.text	Hilfsklassen zur Lokalisierung und Internationalisierung

5.3 Java im Vergleich mit C++

Die Syntax von Java ist sehr eng an C++ angelehnt; an den Unterschieden läßt sich - auch wenn man C/C<++ nicht kennen sollte, recht gut verdeutlichen, was die Besonderheiten von Java sind:

- kein Präprozessor, d.h. u.a. kein Typedef, kein Define
- keine Records und Unions
- keine Funktionen
- keine Mehrfachvererbung bei Klassen
- kein **goto**
- kein Überladen von Operatoren
- keine Templates
- keine Pointer
- Interfaces zum Import anderer Klassen statt #include von Headerdateien
- verschiedene Typen von Kommentaren
- automatische Speicherverwaltung
- synchronisierte threads werden unterstützt

5.4 Sprachbeschreibung von Java

5.4.1 *Zeichenkodierung*

- Kodierung von Zeichen in UNICODE
- Zusätzlich Mapping von ASCII nach UNICODE, um auch Source-Code auf nicht-UNICODE-Systemen zu unterstützen (UNICODE als Obermenge von ASCII)

5.4.2 *Lexikalische Elemente von Java-Programmen*

Quelltext = {Kommentar|Leerzeichen|Sprachelement}
Leerzeichen = ASCII-SP | ASCII-HT | ASCII-FF | Zeilenendezeichen
Sprachelement =Schlüsselwort | Identifikator | Literal | Separator | Operator

5.4.3 *Schlüsselwörter*

abstract	boolean	break	byte	(byvalue)	case	(cast)
catch	char	class	(const)	continue		
default	do	double	else	extends	final	
float	for	(future)	(generic)	(goto)	if	
implements	import	(inner)	instanceof	int	interface	
long	native	new	null	operator	(outer)	
package	private	protected	public	(rest)	return	
short	static	super	switch	sychronized	this	
throw	throws	transient	try	(var)	void	
volatile	while					

Bemerkung: Die Schlüsselwörter byvalue, cast, const, future, generic, goto, inner, operator, outer, rest, var sind reserviert, werden aber noch nicht benutzt.

5.4.4 Programmstruktur

Quelltextdatei = [Paketfestlegung] {Import} {Typdeklaration}

Paketfestlegung = package Paketname;
Paketname = Bezeichner.

Import = Typimport | Paketimport

Typdeklaration = Klassendeklaration | Schnittstellendeklaration

Beispiel:

```
package NeuesPaket;
import java.applet.applet;
class Test extends Applet
{
    int EineEigenschaft = 0;
    int EineMethode(int EinFormalerParameter)
    {
        return EinFormalerParameter + 5;
    }
}
```


5.4.5 Bündelung von Kompilierungseinheiten in Paketen

Einzelne Java-Code-Modulen werden zu sog. Packages zusammengefaßt
Java-Kernsystem: Basis-Packages **applet, util, awt, net, lang** und **io**
Zugriff auf Codemodule auf unterschiedlichen Servern muß möglich sein muß
global unique package names: Internet-Domänenname (COM.sun, DE.gesis etc.) ..., z.B. COM.sun.java.jag.scrabble).
Lokale package-Namen werden in Verzeichnispfade übersetzt, z.B. ergibt **jag.fun-scrabble jag/fun/scrabble**
Jedes Package besteht aus einer Reihe von Compilation Units, durch Import statements werden externe Klassen verfügbar, z.B. **import java.util.Vector** oder **import java.awt.***, d.h. Wildcards sind erlaubt.

5.4.6 Bezeichner

Bezeichner für Variablen, Klassen, Objekte etc. können in Java beliebige Länge aufweisen, das erste Zeichen des Bezeichners muß ein (Unicode-)Buchstabe sein, danach können Buchstaben und Ziffern in beliebiger Anordnung folgen.

Syntax: Bezeichner = UNICODE-Buchstabe {UNICODE-Buchstabe | UNICODE-Ziffer}

Was kann man bezeichnen?

- Pakete (package *Bezeichner*)
- Klassen (... *class Bezeichner*)
- Schnittstellen (... *interface* Bezeichner ...)
- Variablen (Typ Bezeichner [Initialisierung])
- Methodendeklarationen (Rückgabetyp Bezeichner ([Parameterliste])
- Formale Parameter
- Sprungmarken (Bezeicher : Anweisung)

5.4.7 Kommentare

Drei verschiedene Kommentartypen; neben einzeiligen und mehrzeiligen Kommentaren (C++) eine Sonderform für die automatische Dokumentationsgenerierung:

/** Kommentar für automatische Dokumentation*/

5.4.8 Literale

Literale (Konstanten) dienen der Darstellung von konstanten Werten, Zeichen, Zeichenketten (wie C++), zusätzlich Wahrheitswerte **true, false** für den Datentyp **boolean**.

5.4.9 Datentypen

- Einfache bzw. primitive Datentypen
- Referenzdatentypen: Array-Typen, Klassen-Datentypen, Interface-Datentypen

Typ	*interne Speicherung*	*darstellbare Werte*	*Vorbelegung*
byte	8bit	-128 bis 127	0
short	16bit	-32768 bis 32767	0
int	32bit	-2147483648 bis 2147483647	0
long	64bit	-9223372036854775808 bis 9223372036854775807	0
float	32 bit		0.0
double	64 bit		0.0
char	16 bit	\u0000 - \uffff	\u0000 (Nullcharacter)
boolean		true, false	false

5.4.10 Arrays

Arrays sind echte Objekte in Java, nicht lediglich aus den Basisdatentypen abgleitet.
Syntax für eindimensionale Arrays ist der von C vergleichbar:

```
Int IntReihe[];
Int[] IntReihe;
```

Arrays sind in Java immer nur eindimensional. Das bedeutet, daß jeder Array ein Objekt ist, das 0-n Variablen enthält, die Komponenten des Arrays. Alle Komponenten haben den gleichen Typ. Da eine Komponente eines Arrays selbst ein Arrays sein kann, erhält man auf diese Weise die Möglichkeit, mehrdimensionale Arrays zu spezifizieren. Bei mehrdimensionalen Arrays muß der "innerste" selbstverständlich ein Array mit Kompone nten eines primitiven Datentyps sein. Die Variablendeklaration selbst alloziert noch keinen Speicher für den Array, lediglich für die Variable selbst wird Platz geschaffen.
Beispiele für tatsächliche Allokation eines Arrays:

```
Object test4[][] = new int [2][3];
int[] test5 = {1,2,5,6,7,0};
String[]test6 = {"ein", "Array", "aus", "Strings"};
```

5.4.11 Variablen

Variablen sind typisierte Speicherplätze, in denen man Werte ablegen kann (typed storage locations). Jede Variable hat als Attribute ihren Typ und ihre Speicherklasse, wobei der Typ entweder einfach (primitive) ist oder ein Objekttyp (object, interface, array).

Speicherklassen:

- local: innerhalb eines lokalen Anweisungsblocks

- static: lokal bezüglich einer Klasse; die Variable existiert während der Lebenszeit des Objekts, zu dem sie gehört
- dynamic: Instanzen von Objekten und Klassen, alloziert durch new, können von mehr als einer Variable referenziert werden, die automatic garbage collection überprüft jeweils Möglichkeiten der Speicherfreigabe für dynamische Variablen.

5.4.12 Konstante (final Variablen)

In Java gibt es kein besonderes Sprachkonstrukt für Konstante. Stattdessen kann man eine Varaible als *final* kennzeichnen, was bedeutet, daß ihr Wert nicht verändert werden kann.
Beispiel:

```
final float E = 2.7;
```

5.4.13 Anweisungen und Anweisungsblöcke

Blockaufbau und Variablendeklarationen erfolgen im wesentlichen nach den von C++ bekannten Regeln.

Beispiel: Methoden- und Komponentenzugriffe
Auf die Methoden einer Klasse (eines Objektes einer Klasse) kann man unter Angabe von Klase und Methode zugreifen:

```
Objektname.Methodenname();
```

Z.B.:

```
System.out.println();
```

Gleiches gilt für die Komponenten eines Objekts:

```
MeinObjekt.MeineEigenschaft = 0;
```


5.4.14 Operatoren und Separatoren

verfügbare Operatoren im wesentlichen mit den aus C/C++ bekannten Operatoren identisch:

Separatoren:

```
()   []   {}   ;   ,   .
```

Operatoren:

```
=    >    <    !    ?    :    ==   <=   >=   !=   &&   ||
++   --   +    -    *    /    &    |    ^    %    <<   >>
>>>  +=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=
>>>=
```

Wichtig: Der Additionsoperator ist auch für Objekte vom Typ String definiert (string concatenation).

5.4.15 Kontrollflußsteuerung

Im wesentlichen identisch mit C++ (**if, switch, while. do/while, for**). Nachfolgend nur Abweichungen.

Sprunganweisungen
Der Kontrollfluß kann auch unbedingt durch Sprunganweisungen gesteuert werden:

break [Bezeichner] - Verlassen einer Ausführungsumgebung
continue [Bezeichner] - Fortsetzen einer umgebenden Anweisung
return [<Ausdruck>] Wertrückgabe

Durch ein Break mit Label ist ein sog. “Mehrebenen-Break” möglich, bei dem nicht nur die nächste Schleifenumgebung verlassen wird.
Beispiel Break mit Label:


```
int i = 10, j = 0, k = 0;
Ziel_des_Sprungs: while(i > 0)
{
      do
      {
         j++;
         for(k = 5; k >= 0; k--)
         {
               if(k == 3)
               break Ziel_des_Sprungs;
         }
      }while(j < 10);
      i--;
}
System.out.println(k);
```

Ausgabe:

```
3
```

5.4.16 Defensives Programmieren mit Ausnahmen

Möglichst frühzeitig sollen vom "normalen" Verhalten eines Programms abweichende Ereignisse abgefangen werden. Gerade bei Zugriff auf externe Ressourcen (Dateien, Netzwerkverbindungen etc.) tritt häufig der Fall ein, daß etwa ein Dateizugriff nicht gelingt da die Datei gesperrt ist oder eine Netzwerkressource nicht erreicht werden kann.
Dazu dienen in Java Ausnahmen. Ausnahmen können durch eine **throws**-Anweisung einer Methode zugeordnet werden. in Java gibt es eine Reihe vordefinierter Ausnahmen. Eine Ausnahme wird durch die

throw-Anweisung explizit ausgelöst

Beispiel: Ausnahmeauslösung:

```
static int teile(int x, int y) throws ArithmeticException
{
      if(y==0)
      throw(new ArithmeticException("0-Division"));
      else
      return x / y;
}
```

Abarbeiten von Ausnahmen

try-, **catch**- und **finally**-Anweisungen dienen der Bearbeitung von Ausnahmen

Folgende Varianten sind erlaubt:
- try <Anweisungsblock> <catch-Block>
- try <Anweisungsblock> [<catch-Block>] finally

Ein mit **try** umschlossener Anweisungsblock muß einen **catch**-und/oder **finally**-Block nach sich ziehen.

Beispiel try-/catch-Anweisung

```
class TryCatch
{
    public static void main(String[] args)
    {
        try
        {
            int x = teile(1,0);
        }
        catch (Exception e)
        {
            System.out.println("Ausnahme:" +
            e.getMessage());
        }
    }
    static int teile(int a, int b) throws Exception
    {       return a / b; }
}
```


5.5 Objektorientierung in Java

5.5.1 OO-Basismodell in Java

Klassen:

- Deklaration
- Eigenschaften
- Methoden

5.5.2 Die Klassendeklaration

Eine Klassendeklaration führt einen *Referenztyp* ein, dessen Implementierung von der Implementierung einer anderen Klassen abgeleitet sein kann. Jede Klasse kann unmittelbare oder mittelbare *Superklasse* einer anderen sein.

Aufbau der Klassendeklaration:

[public]
[Klassenmodifikator (abstract, final)]
class Bezeichner
[extends Klassentyp]
[implements Schnittstellentyp {, Schnittstellentyp}]
Klassenrumpf

Beispiele:

```
class Test0 {...}
public class Test1 {...}
abstract class Test2 {...}
public final class Test4 extends SuperTest implements InterfaceExmpl {...}
```


5.5.3 Der Klassenrumpf

Klassenrumpf = { Komponentendeklaration | Initialisierung }
Der Klassenrumpf besteht aus einer oder mehrerer Felddeklarationen, d.h. Variablen- und Methoden-vereinbarungen sowie aus Initialisierungen:

```
public class test
{
    int test;           //Deklaration
    int test2 = 0; //Deklaration und Initialisierung
    {
        test = 5;       // Initialisierung in separatem Block
    }
    int pow2(int test)
    {
        return test * test;
    }
}
```

5.5.4 Sichtbarkeit von Klassen

Das Schlüsselwort **public** in der Klassendeklaration einer (Toplevel-) Klasse regelt deren Sichtbarkeit: Ist die Klasse **public**, ist sie auch außerhalb ihres Paketes sichtbar, ansonsten ist sie nur in dem Paket sichtbar, in dem sie deklariert wurde.

5.5.5 Erzeugen von Objekten

Unterschied Deklaration und Erzeugung:

```
class Seite
{
        String Inhalt;
}
class Buch
{
        Seite Index, Register;
        Index = new Seite();
        Register = new Seite();
}
```

5.5.6 Methoden und Eigenschaften (Felder)

Die Methodendeklaration - vollständige Syntax:

[public | protected | private	(Sichtbarkeit der Methode)
[static]	(Zuordnung Klasse/Objekt)
[abstract | final]	(Abstraktionsgrad/Vererbbarkeit)
[native]	(Zugriff auf Plattformspezifisches)
[synchronized]	(Synchronisation von threads)

Resulttattyp Methodenname {[]} ([ÜbergabeParameter]{, ÜbergabeParameter})
[Ausnahmen] Methodenrumpf

"Mindestens" muß eine Methodendeklaration also wie folgt aussehen:

```
Resulttattyp MethodenName() { //Methodenrumpf }
```

z.B.

```
void MeineMethode()
{
    int x = 5;
}
```

Parameterübergabe: Die Parameterübergabe erfolgt als *call by value*, d.h. der Wert des formalen Parameters und nicht eine Referenz auf ihn wird an die Methode übergeben.

Beispiel:

```
public class CallByValue
{
    public static void main(String argv[])
    {
        int eineZahl = 100;
        System.out.println("Vor dem Aufruf:" + eineZahl + "\n");
        Aufruf(eineZahl);
        System.out.println("Nach dem Aufruf:"+eineZahl+"\n");
    }
    public static void Aufruf(int dieZahl)
    {
        dieZahl*=dieZahl;System.out.println("In der Methode:"+dieZahl+ "\n");
    }
}
```


Zu beachten ist, daß bei der Übergabe von Objektreferenzen (d.h. einer Variablen, die einer Referenz auf ein Objekt entspricht), die Eigenschaften des Objekts modifiziert werden können, d.h. "an sich" einem *call by reference* entsprechen. Die als Parameter übergebene Objektreferenz bleibt natürlich unverändert (*call by value*).

5.5.7 Polymorphismus (Überladen)

Eine Methode kann unter gleichem Namen mehrfach in einer Klasse deklariert werden; so erhält man Flexibilität hinsichtlich der Behandlung unterschiedlicher Übergabeparameter.

Beispiel (fortführend das obige call by value-Beispiel):

```
public static void Aufruf(int dieZahl)
{
    dieZahl *= dieZahl;
    System.out.println("In der Methode:" + dieZahl + "\n");
}
public static void Aufruf(String derString)
{
    derString = derString + derString + derString;
    System.out.println(derString);
}
```

Hinweis: Man beachte den Unterschied zwischen *overloading* und *overriding*, das bei dem Überschreiben einer Oberklassenmethode durch deren "Reimplementierung" in einer Unterklasse auftritt.

5.5.8. Konstruktoren

Konstrukten sind ein Sonderfall von Methoden: Mit Ihnen werden Objekte einer Klasse erzeugt. Ein Konstruktor einer Klasse hat denselben Bezeichner wie die Klasse selbst. Für eine Klasse kann es mehrere (überladene) Konstruktoren mit unterschiedlichen Parametern geben (Polymorphismus).

Beispiel: Für die vordefinierte Klasse **java.lang.String** existieren sieben verschiedene Konstruktoren:

```
public String()
public String(String value)
public String(char[] value)
public String(char[] value, int offset, int count)
public String(byte[] ascii, int hibyte, int offset, int count)
public String(byte[] ascii, int hibyte)
public String(StringBuffer, buffer)
```

Besonderheiten: Stehen mehrere Konstruktoren zur Verfügung, so kann aus einem Konstruktor einer Klasse ein anderer Konstruktor mit Hilfe von this() ausgelöst werden. In diesem Fall werden zwei Konstruktoren verarbeitet.

Beispiel:

```
class Konstrukt
{
    String Name;int Status;
    void Konstrukt(int S)
    {       Status = S;}
    void Konstrukt (String N, int S)
    {   this(S);Name = N;}
}
```


Auslösereihenfolge für Konstruktoren:

- Konstruktor der Oberklasse (so vorhanden)
- Initialisierung der Datenfelder der Klasse
- Ausführung des Konstruktorrumpfs

5.5.9 Klassenbezogene Methoden und Eigenschaften

Methoden wie Eigenschaften können klassen- oder objektbezogen existieren, wobei per default jede Methode oder Eigenschaft einer Klasse objektbezogen ist. Durch das Schlüsselwort static werden Methoden und Eigenschaft klassenbezogen.

Beispiel:

```
class Auto
{
    public static int AnzahlDerRäder;
    public static void SetAnzahlDerRäder(int iNum)
    {
        AnzahlDerRäder = iNum;
    }
}
```

```
public class Statisch
{
      public static void main(String argv[])
      {
         Auto VW, StretchLimo;
         VW = new Auto();StretchLimo = new Auto();
         VW.AnzahlDerRäder = 4;System.out.println(VW.AnzahlDerRäder + "\n");
         StretchLimo.AnzahlDerRäder = 6;
         System.out.println(StretchLimo.AnzahlDerRäder + "\n");
         System.out.println(VW.AnzahlDerRäder + "\n");
         System.out.println(StretchLimo.AnzahlDerRäder + "\n");
         System.out.println("Ein letztes Mal:\n");
         VW.SetAnzahlDerRäder(123456);
         System.out.println(VW.AnzahlDerRäder + "\n");
         System.out.println(StretchLimo.AnzahlDerRäder + "\n");
      }
}
```


5.5.10 Vererbung

In Java existiert zwischen Klassen nur die Einfachvererbung, d.h. eine Klasse kann immer nur von einer Oberklasse abgeleitet sein:

```
class Unterklasse extends Oberklasse
{
      //...
}
```

Mehrfachvererbung ist nur indirekt, d.h. über die Implementierung einer Schnittstelle durch eine Klasse möglich (s.u.).

5.5.11 Zugriffskontrolle von Methoden und Eigenschaften

Die Zugriffsmöglichkeiten einer Unterklasse auf Methoden und Eigenschaften von Oberklassen wird durch Zugriffsmodifikatoren geregelt:

- **public**: Methode kann von anderen Klassen benutzt werden, auch außerhalb der Klassenhierarchie
- **protected**: Methode kann nur innerhalb der Klassenhierarchie benutzt werden
- **private:** Methode kann nur von Objekten der Klasse, in der sie definiert ist, benutzt werden, wird also auch nicht vererbt

Beispiel:

```
public class Vererbung
{
   public static void main(String argv[])
   {
         Oberklasse OK = new Oberklasse(); Unterklasse UK = new Unterklasse();
         OK.tuwasOK();UK.tuwasUK();UK.tuwasOK();
   }
}
class Oberklasse
{
   private void tuwasnurinOK(){System.out.println("tuwasnurinOK\n");}
   protected void tuwasOK(){tuwasnurinOK();System.out.println("tuwasOK\n");}
}
class Unterklasse extends Oberklasse
{
   protected void tuwasUK()
   {
         // super.tuwasnurinOK(); nicht möglich, da private
         // tuwasnurinOK(); nicht möglich, da private
         tuwasOK(); // möglich, da protected
         System.out.println("tuwasUK\n");
   }
}
```


5.5.12 finale und abstrakte Klassen

Von einer Klasse, die als final gekennzeichnet ist, kann man keine Unterklasse ableiten. Umgekehrt muß man von einer als abstract gekennzeichenten Klasse eine Unterklasse ableiten, um sie verwenden zu können. Eine Unterklasse einer abstrakten Oberklasse muß alle dort vorgegebenen abstrakten Methoden implementieren, sonst wird sie selbst zur abstrakten Klasse.

Beispiel:

```
public class FinalAbstrakt
{
      public static void main(String argv[])
      {
      //Oberklasse OK=new Oberklasse();
      //Fehler, abstrakte Klasse
         FinaleUnterklasse UK = new
         FinaleUnterklasse();
         UK.tuwasUK();
         UK.tuwasOK();
      }
}
```

```
abstract class AbstrakteOberklasse
{
	private void tuwasnurinOK()
	{System.out.println("tuwasnurinOK\n");}
	final protected void tuwasOK()
	{tuwasnurinOK();System.out.println("tuwasOK\n");}
	abstract void Untenzuimplementieren();//nur Methodenkopf
}
final class FinaleUnterklasse extends AbstrakteOberklasse
{
	//protected void tuwasOK(){};geht nicht, final in Oberklasse
	void Untenzuimplementieren()
	{System.out.println("s.o.: abstrakte Methode\n");};
	protected void tuwasUK()
	{
	   //super.tuwasnurinOK(); nicht möglich, da private
	   //tuwasnurinOK(); nicht möglich, da private
	   tuwasOK();    System.out.println("tuwasUK\n");
	}
}
//	public class NochEineDrunter extends FinaleUnterklasse { }
//	geht nicht, da final
```


5.5.13 Schnittstellen

Mit einer Schnittstelle gibt man eine Menge von Methoden und Eigenschaften vor, die von einer Klasse implementiert werden sollen. Alle Methoden einer Schnittstelle sind abstrakt, d.h. ohne Implementierung Eine Schnittstelle liefert also eine Funktionsschablone, die von unterschiedlichen Klassen implementiert werden kann. Eine Klasse kann auch mehr als eine Schnittstelle implementieren.

Syntax:
[public] [abstract] interface *Schnittstellenname*
[extends Schnittstellenname {, Schnittstellenname}]
Schnittstellenrumpf

Beispiel:

```
interface Darstellbar
{
	int Groesse = 2000; int Flaeche = 500;
	void StelleDar();
}
```



```
public class Schnittstelle
{
	public static void main(String argv[])
	{
	   Gegenstand Irgendwas = new Gegenstand();
	   Irgendwas.StelleDar();
	}
}
class Gegenstand implements Darstellbar
{
	public void StelleDar()
	{
		System.out.println(Groesse+", "+Flaeche);
		// Groesse = 1234567;geht nicht, da "automatisch" final
	}
}

//	class Ding implements Darstellbar
//	{ “leerer Rumpf”} geht nicht, da keine Implementierung von StelleDar()
// aber - als abstrakte Klasse möglich:
abstract class Etwas implements Darstellbar{}
```


5.5.14 Methoden und Variablen in Schnittstellen

Alle Methoden einer Schnittstelle sind (implizit) abstrakt, müssen also nicht also solche gekennzeichnet werden; sie sind öffentlich (public), falls die Schnittstelle öffentlich ist. Die Eigenschaften einer Schnittstelle sind immer statisch und final dienen zur Bestimmung von Konstantenwerten, d.h. sie müssen im Schnittstellenrumpf initialisiert werden. Ihr Wert läßt sich nicht ändern. Die Schnittstellensyntax zeigt, daß eine Schnittstelle von mehreren anderen Schnittstellen abgeleitet sein kann; auf dieser Ebene ist Mehrfachvererbung möglich (*diamond inheritance*).

5.5.15 Innere Klassen

Seit dem JDK 1.1.X existieren als neues Konzept sog. "innere Klassen" (und Schnittstellen), d.h. Klassen, die innerhalb einer anderen Klasse deklariert werden (im Gegensatz zu den bisher allein verfügbaren "Toplevel"-Klassen). Innere Klassen sind damit ein weiterer Fall einer Komponente, die innerhalb einer Klasse enthalten sein kann.
Man unterscheidet

- innere Klassen
- lokale Klassen
- anonyme Klassen

Innere Klassen - Beispiel:

```
public class TopLevelKlasse
{
        public static void main(String[] args)
        {new InnereKlasse1();}
        static class InnereKlasse2
        {
            InnereKlasse2(){System.out.println("Konstruktor InnereKlasse2");}
            static {System.out.println("Initialisierung InnereKlasse2");}
        }
        static class InnereKlasse1
        {
            InnereKlasse2 meineZeite = new InnereKlasse2();
            InnereKlasse1(){System.out.println("Konstruktor InnereKlasse1");}
            static {System.out.println("Initialisierung InnereKlasse1");}
        }
}
```

Ausgabe:

```
Initialisierung InnereKlasse1
Initialisierung InnereKlasse2
Konstruktor InnereKlasse2
Konstruktor InnereKlasse1
```


5.6 Programmtypen in Java: Applets und Applications

Es gibt drei Typen Java-basierter Programme:

- ***applications:*** stand alone Java-Programme
- ***applets***: in HTML-Seiten integrierte Java-Programme
- ***servlets:*** Java-Programme, die auf einem (Java-basierten) Web-Server laufen.

5.6.1 Besonderheiten von Applications

Ausführung durch den Javainterpreter *java*
main-Methode als Programmeintrittspunkt für den Interpreter:
public static void main(String[] args) {...}
Ausgabe an Konsole: System.out.println
Aufruf mit Kommandozeilenargumenten möglich: Auswerten des Stringarrays (**String[] args**) der **main**-Methode. In **args[0]** steht das erste Kommandozeilenargument (also diejenige Zeichenfolge, die in der Kommandozeile auf den Programmnamen folgt - Unterschied zu C/C++)

5.6.2 Appletcodierung in HTML

```
<html><head><title>Simple</title></head>
<body>
<applet code=Simple.class width=400 height=400>
</applet>
</body></html>
```


Vollständige Syntax des APPLET-Tags:

```
<APPLET
CODE = appletFile | 'OBJECT' = objectfile | 'ARCHIVE' = JarFile
WIDTH = pixels
HEIGHT = pixels
[CODEBASE = codebaseURL]
[ALT = alternateText]
[NAME = appletInstanceName]
[ALIGN = alignment]
[VSPACE = pixels]
[HSPACE = pixels]
>

[<PARAM NAME = appletAttribute1 VALUE = value>]
[<PARAM NAME = appletAttribute2 VALUE = value>]
...
[alternateHTML]
</APPLET>
```


5.6.3 Struktur von Applets

Anders als ein Standard-Programm verfügt ein Applet nicht über eine main()-Methode - es muß von der Klasse Applet abgeleitet und sein Klassennamemuß identisch mit dem Dateinamen sein. Die Standard-methoden der Klasse Applet können nach Bedarf angepaßt werden

Beim “Betreten" einer HTML-Seite, die ein Applet enthält, wird das Applet geladen, eine Instanz seiner Klasse erzeugt und das Applet gestartet. Das Applet läuft dann (“Applet running" in der Statuszeile), bis die Seite wieder verlassen wird. Beim verlassen wird es angehalten und die Instanz abschließend zer-stört. Diesen einzelnen Schritten entsprechen die folgenden Standardmethoden der Klasse Applet:

```
start()
init()
stop()
destroy()
```

Beispiel eines Applets:

```
import java.applet.*;import java.awt.*;
public class Simple extends Applet
{
        public void paint(Graphics g)
        {
            g.drawRect(0, 0, size().width-1,size().height-1);
            g.drawString(“Hello, World”);
        }
}
```

5.6.3 Eigenschaften der Applet-Klasse

Die Klasse Applet ist abgeleitet von Object, Component, Container und Panel, wodurch folgende Eigenschaften gewährleistet werden:

- Allgemeine Eigenschaften von Java-Objekten (Basisklasse Object)
- Erscheinen als visuelles Element auf dem Bildschirm (Component)
- Fähigkeit, andere visuelle Elemente zu beinhalten (Container)
- Fähigkeit, die gekapselten visuellen Elemente des Applets auch auszurichten (Panel)

5.6.4 Ausgewählte Methoden java.applet.Applet

public boolean isActive()
Prüft, ob das Applet aktiv ist. Ein Applet wird als aktiv markiert, bevor seine start()-Methode aufgerufen wird.

public URL getDocumentBase()
Liefert die URL des Dokumentes, in das das Applet eingebettet ist.

public URL getCodeBase()
Liefert die Basis-URL. Dies ist die URL des Applets selbst.

public void showStatus(String msg)
Zeigt eine Statusmeldung im Kontext des Applets (Beispiel in ImageMap). Die Ausgabe erfolgt z.B. in der Statuszeile von Netscape.

public Image getImage(URL url)
Lädt ein Bild von der gegebenen URL.
public AudioClip getAudioClip(URL url)

5.7 Die Entwicklungsumgebung des Java Development Toolkit

- Von SUN frei verfügbare Standardumgebung, umfaßt Classcompiler, Interpreter, Appletviewer und einfachen Debugger.
- Ausgangspunkt der meisten Implementierungen
- kommandozeilenorientiert
- für alle Javaplattformen verfügbar

Im Lieferumfang des JDK befinden sich eine Reihe weiterer Werkzeuge zur Applikationsentwicklung sowie zur automatischen Dokumentationsgenerierung.

5.7.1 Betrachten von Applets

Die naheliegende Möglichkeit, ein Applet zu betrachten, besteht darin, es in einem Java-fähigen Browser zu laden, eingebettet in eine HTML-Seite. Um diese laufen zu lassen, ist wiederum auf PCs Windows95 oder WindowsNT nötig; weitere Plattformen sind SUN Solaris, Macintosh und weitere UNIX-Derivate. Eine weitere einfache Alternative ist der mit dem JDK gelieferte **Appletviewer**, der ein Applet gewissermaßen als stand-alone-Anwendung startet und zum schnellen Testen geeignet ist. Auch hier muß das Applet allerdings in eine HTML-Seite eingebettet sein.

Aufruf:
appletviewer HTMLDateiname.html

5.7.2 Compiler

Der Compiler **javac** dient dazu, aus dem Quelltext (Datei mit Extension .java) eine Klassendatei (.class) zu kompilieren. Diese wird dann auf der jeweiligen Plattform interpretiert und ausgeführt. Um ein Applet laufen lassen zu können, benötigt man also seine class-Datei. Diese wird über das Netz übertragen,. als Bytecodefolge interpretiert und dann im Browser oder von einem Appletviewer ausgeführt.

Aufruf:
javac Dateiname.java

Einige Optionen:
-d Ablageverzeichnis für Classdateien
-g Informationen zur Fehlersuche
-classpath Pfadangabe Liste von Suchpfaden (Pfad;Pfad) für Klassenbibliotheken
-verbose Angabe aller eingebundenen Klassen

5.7.3 Interpreter

Der Standard Java-Interpreter wird mit “java” und einem Klassennamen als Argument von der Kommandozeile aus aufgerufen. Mit ihm können “Standard-Javaprogramme” (applications) ausgeführt werden.

Aufruf:
java Klassenname

Optionen:
-classpath, -verify, verbose etc.

5.7.5 Dokumentationserstellung

Das Programm **javadoc** erlaubt die automatische Erstellung von Dokumentation im HTML-Format auf der Basis der *.java Quelltexte. Gibt man für javadoc als Kommandozeilenargumente alle relevanten Packages und Verzeichnisse an, so wird nicht nur für jede Quelldatei ein eigenes Dokumentationsfile erstellt, sondern darüber hinaus auch eine Indexdatei und ein Klassenbaum erstellt. Die einzelnen Daten sind dann über Hypertextlinks untereinander vernetzt und können als zusammengehöriges Hilfesystem benutzt werden.